05-25-06

IFW

CERTIFICATE OF MAILING BY "EXPRESS MAIL" (37 CFR 1.10) Applicant(s): **Raman et al.**			Docket No. **ARC920030007US1**	
Application No. **10/645,221**	Filing Date **August 21, 2003**	Examiner **Kimberly M. Lovel**	Customer No. **29154**	Group Art Unit **2167**

Invention:
SYSTEM AND METHOD FOR ASYNCHRONOUS DATA RELICATION WITHOUT PERSISTENCE FOR DISTRIBUTED COMPUTING

I hereby certify that this **Declaration Under 37 C.F.R. 1.131**
(Identify type of correspondence)

is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 in an envelope addressed to: Director of the United States Patent and Trademark Office, P.O. Box 1450, Alexandria, VA 22313-1450 on **May 23, 2003**
(Date)

Tylene McCoy
(Typed or Printed Name of Person Mailing Correspondence)

(Signature of Person Mailing Correspondence)

EV 815252254 US
("Express Mail" Mailing Label Number)

Note: Each paper must have its own certificate of mailing.

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re patent application of:
Raman et al.

Serial No.: 10/645,221

Filed: August 21, 2001

Group Art Unit: 2167

Examiner: Lovel, Kimberly M.

Atty. Docket No.: ARC920030007US1

For: SYSTEM AND METHOD FOR ASYNCHRONOUS DATA REPLICATION WITHOUT PERSISTENCE FOR DISTRIBUTED COMPUTING

Commissioner of Patents
P.O. BOX 1450
Alexandria, VA 22313-1450

DECLARATION UNDER 37 C.F.R. §1.131

We, the inventors of the invention defined by claims 1-52 of U.S. Patent Application Serial No. 10/645,221 hereby declare the following:

[0001] The purpose of this declaration is to prove that we conceived the claimed invention prior to the earliest effective prior art date of portions of U.S. Patent No. 6,889,231 issued to Souder et al., which is presently understood to be September 13, 2002. The following shows that we conceived our invention prior to September 13, 2002 and that we were diligent from our date of conception to its reduction to practice and were further diligent to the date of the filing of our patent application, which has a filing date of August 21, 2003 (hereinafter, the

"Patent Application").

[0002] We are the only inventors of the subject matter claimed in claims 1-52 of U.S. Patent Application Serial No. 10/645,221.

[0003] During all time periods mentioned herein, and specifically between our conception date and the filing date of the application, all activities described herein occurred in the United States.

[0004] Proof of the conception of the claimed invention prior to September 13, 2002, and diligence in reducing the invention to practice and filing the Patent Application is demonstrated in the attached Exhibits, labeled as Exhibits A through B.

[0005] As shown in Exhibit A, which is a product data sheet, we conceived the claimed invention at a date prior to September 13, 2002. As permitted by MPEP §715.07, the dates on Exhibit A have been removed; however, we hereby declare that all relevant dates thereon are prior to September 13, 2002. Further, the invention was actually conceived before Exhibit A was prepared. Therefore, our conception date actually predates Exhibit A.

[0006] Exhibit A refers to an "attached paper" (see page 2 of Exhibit A). This "attached paper" is provided in Exhibit B. As shown in Exhibit B, which is a company white paper, we conceived the claimed invention at a date prior to September 13, 2002. As permitted by MPEP §715.07, the dates on Exhibit B have been removed; however, we hereby declare that all relevant dates thereon are prior to September 13, 2002. Further, the invention was actually conceived before Exhibit B was prepared. Therefore, our conception date actually predates Exhibit B.

[0007] Exhibits A and B specifically disclose the claimed invention. Specifically, the claimed invention is generally described on pages 1-2 of Exhibit A and pages 1-7 of Exhibit B, and in particular, is provided in the software code provided on pages 3-6 of Exhibit B. The

features provided in claims 2-17, 19-34, and 36-51 are generally inferred in Exhibits A and B. For example, amended independent claims 1, 18, 35, and 52 generally define assigning a delta production/consumption value for arbitrary data sources and targets operable for replicating data (see generally Exhibit B, page 1); embedding replication tracking information within said data (see generally Exhibit B, pages 5-6), wherein said replication tracking information comprises a timestamp and a contiguous sequence number (see generally Exhibit A, page 2); atomically and independently applying updates at a target site using said replication tracking information (see generally Exhibit A, page 2); and using an apply service at said target site to embed and analyze said tracking information during a crash recovery sequence (see generally Exhibit A, page 2; Exhibit B, pages 4-6), wherein said apply service utilizes an in-memory index when a system crash occurs and a recovery process is initiated by said distributed computing system (see generally Exhibit B, pages 4-6).

[0008] We were diligent from the date of conception in reducing the claimed invention to practice and in pursuing, preparing, and filing the Patent Application. More specifically, on July 8, 2002, information similar to that shown in Exhibits A and B was presented to a patent attorney/agent to determine whether a patent application should be prepared.

[0009] Generally, the invention was conceived on or about March 22, 2002 and was reduced to practice on or about July 1, 2002. An exhaustive series of experiments were conducted on the invention testing its validity from March 2002 to July 2002. The testing was quite rigorous and required substantial time, money, and effort to undertake. The results of the experiments were positive, which further resolved the decision to seek patent protection. After the invention was conceived and reduced to practice, and the testing yielded positive results, the decision was reached to seek patent protection due to the potential commercial value and prestige

afforded by the claimed invention. Between October 21, 2002 and December 2, 2002, a prior art search was conducted and analyzed. On January 26, 2003, a further analysis of the prior art was conducted. On February 25, 2003, a patent attorney was instructed to prepare a patent application that eventually became the Patent Application. The Patent Application was eventually prepared and filed on August 21, 2003.

[0010] The foregoing declarations are made according to our best recollection upon review of the appropriate documents and notes, and we hereby acknowledge that willful false statements and the like are punishable by fine or imprisonment, or both (18 USC §1001) and may jeopardize the validity of the application or any patent issuing thereon. All statements made herein are made of our own knowledge and are true and all statements that are made on information and belief are believed to be true.

Bruce G. Lindsay — Date: May 17, 2006

Inderpal S. Narang — Date: May 12, 2006

Vijayshankar Raman — Date: May 11, 2006

Disclosure ARC8-2002-0066

Prepared for and/or by an IBM Attorney - IBM Confidential

Created By Vijayshankar Raman On [redacted] 02:26:11 PM MDT
Last Modified By Cheryl Ruby On [redacted] 05:49:50 PM EST

Required fields are marked with the asterisk (*) and must be filled in to complete the form .

***Title of disclosure (in English)**
Fast replication without 2-phase commits

Summary

Status		Final Decision (File)
Final Deadline		
Final Deadline Reason		
Docket Family		ARC9-2003-0007
*Processing Location		Almaden
*Functional Area	select	(DPB) DPB - Datalinks, SOA & e-commerce (A.Jhingran)
Attorney/Patent Professional		Marc D McSwain/Almaden/IBM
IDT Team	select	Marc D McSwain/Almaden/IBM Inderpal Narang/Almaden/IBM Anant Jhingran/Almaden/IBM
Submitted Date		[redacted] 04:48:30 PM MDT
*Owning Division	select	RES
Incentive Program		
Lab		
*Technology Code		
PVT Score		

Inventors with a Blue Pages entry

Inventors: Vijayshankar Raman/Almaden/IBM, Inderpal Narang/Almaden/IBM, Bruce Lindsay/Almaden/IBM

Inventor Name	Inventor Serial	Div/Dept	Inventor Phone	Manager Name
> Raman, Vijayshankar	0A1036	22/DPBE	457-1110	Narang, Inderpal S.
Narang, Inderpal S.	300147	22/DPBA	457-1743	Jhingran, Anant D.
Lindsay, Bruce G.	929371	22/K01B	457-1747	Morris, Robert J.T.

> denotes primary contact

Inventors without a Blue Pages entry

IDT Selection

***Main Idea**

1. Describe your invention, stating the problem solved (if appropriate), and indicating the advantages of using the invention.

A fast and low overhead asynchronous data replication algorithm. This algorithm avoids persistent queues, persistent change tables, and 2-phase commits that are traditionally needed for asynchronous

replication. The only I/O overhead it imposes is an extra insert statement piggybacked with each transaction that is applied at the target. No persistent state is needed at the source or the target other than the source or target databases themselves.

2. How does the invention solve the problem or achieve an advantage,(a description of "the invention", including figures inline as appropriate)?
The main idea is that the target of replication itself will track the progress of replication, by maintaining a separate table of applied transactions. This table is updated atomically with the application of transactional updates at the target. Each entry in this table has the commit timestamp of the transaction, and a sequence number for the transaction. The commit timestamp is used to check if a given transaction has already been applied, and the sequence number is used to check if any transaction has been lost. The sequence number is assigned by the replication capture program.

When the replication target crashes and recovers, the apply program at the target reads this transaction table and figures out the last contiguous transaction that was committed. The commit timestamp of this transaction is sent to the replication source so that it can start sending updates from there on. A crash at the replication source is treated similarly -- the targets timeout and restart the source, and then send their commit timestamps. In order to reduce the amount of log reading needed during crash recovery, the replication source periodically sends a heartbeat to a replication monitor. This heartbeat has the current log read position and the timestamp of the last committed transaction in the log upto that point. By looking at these heartbeats the replication monitor can estimate how far back the log reader must be moved to read transactions for a particular target.

More details about this are in the attached paper.

repl.pdf

3. If the same advantage or problem has been identified by others (inside/outside IBM), how have those others solved it and does your solution differ and why is it better?
The current IBM replication solutions are of two main kinds:
-- Using a change table: Changes read from the log are entered into a persistent change table. An apply program then reads this table to apply changes to targets. The problem is that this approach reduces throughput because all changes have to go through this persistent table.
-- Using persistent queues: Changes read from the log are directly sent to the target via persistent MQs. This approach needs extra inserts and deletes into the MQ, and also a two-phase commit at the target for removing an update from the MQ and atomically applying it to the target.

4. If the invention is implemented in a product or prototype, include technical details, purpose, disclosure details to others and the date of that implementation.

***Critical Questions (Questions 1-9 must be answered in English)**
***Patent Value Tool (Optional - this may be used by the inventor and attorney to assist with the eval**
Evaluation
Search Information
Search Office Information
Final Decision
Post Disclosure Text & Drawings

Form Revised [redacted])

Efficient Grid Data Replication

[redacted]

1 Goal

We want to design a data replication service for generic grid data sources. The data source is modeled as a *delta producer* which produces self-describing deltas. The replication service understands these deltas only to a very limited extent (see below), and transfers them to the target which is modeled as a *delta consumer*.

The replication service takes as input *replication subscriptions*. Each subscription is a replication request between a single source and single target. Multiple subscriptions can exist to a single source, and can be dynamically added and lost (*e.g.*, when the target or the network fails). A long term goal is to provide guaranteed quality of service, in terms of tolerated latency between source and target.

We want our replication service to impose minimal requirements and overhead on the source and target. Therefore in our design, the source and target of replication do not have any persistent state, other than the database state itself. In particular they do not have persistent change tables, persistent queues, etc. The only I/O overhead imposed by our architecture is an extra insert statement piggy-backed at the end of each transaction applied at the target, into a table with 2 integer columns and no index. We do not do any 2-phase commits.

2 Architecture

The components of the replication service are shown in Figure 1.

The *Monitor service* accepts subscriptions initially, and is responsible for monitoring and maintaining quality of service and initiating crash recovery. The monitor service maintains information about subscriptions in a persistent table to ensure correct behavior across system crashes.

A *Capture service* runs at every replication source. It accepts and buffers deltas from the source's capture program, and is responsible for *flow control* – when the targets can apply deltas at varying speeds, slower than the source can produce them.

An *Apply service* runs at every replication target, accepting deltas and sending them to the target's apply program. The apply service and the target data source are together responsible for tracking the progress of replication, *i.e.*, which deltas have been applied and which have not. When a subscription is restarted after recovery from a crash, this progress information is used to (a) minimize the number of deltas the capture service has to re-send, and (b) avoid applying any delta twice.

Transformations and filters that are common to all subscriptions are performed at the capture side, and others are performed at the apply side.

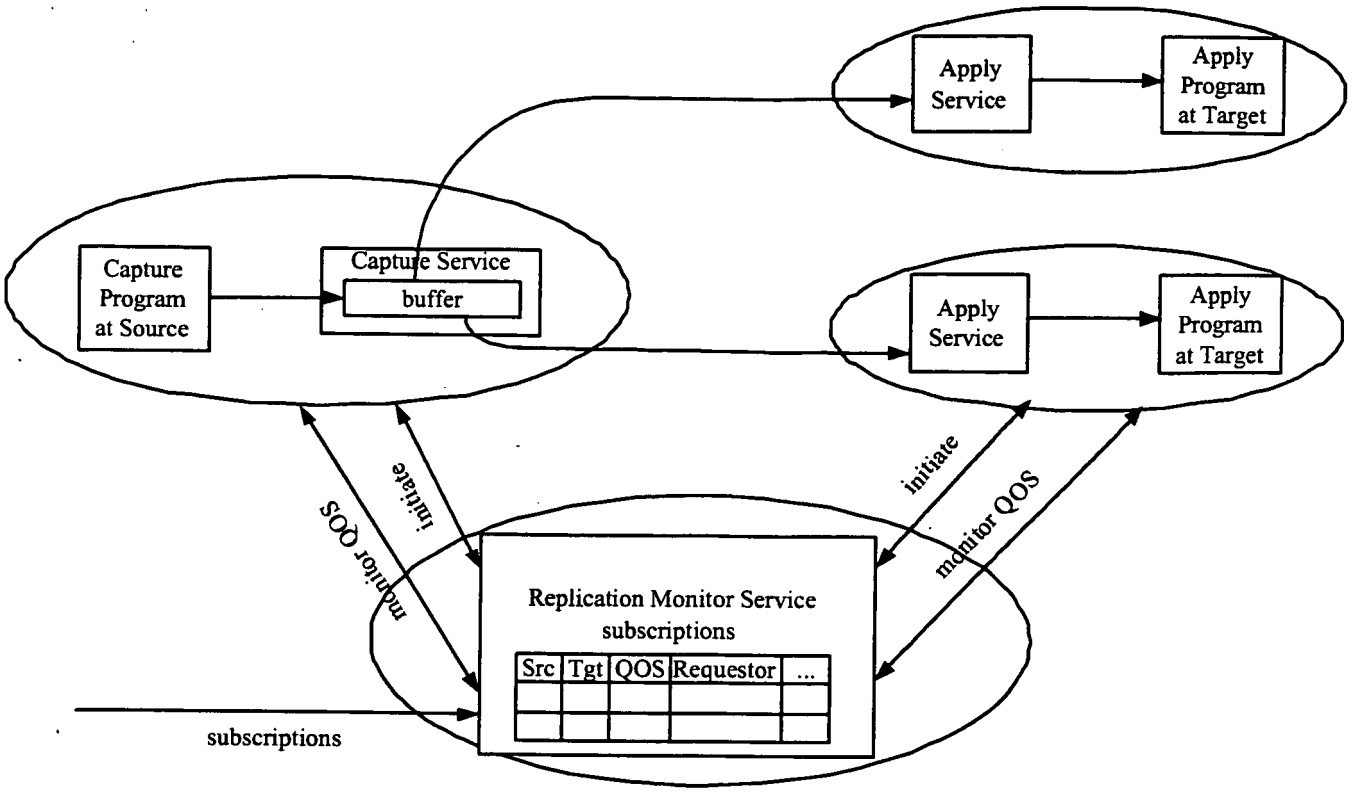


Figure 1: Replication Architecture

3 Replication Algorithm

3.1 No Update-Anywhere, Single Source to Single Target

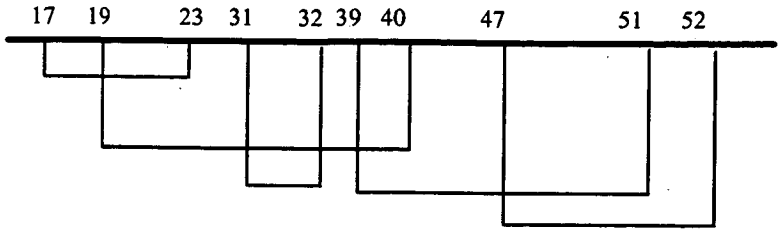
We start with this simple case and generalize to single source to multiple targets in Section 3.2.

3.1.1 Capture Service

The capture service accepts transactional deltas from the source's capture program and sends them to the target. Each δ contains all changes for a specific transaction, and is accompanied by a "time stamp" $commitTS$. $commitTS$ need not be a real timestamp; it can be any number that monotonically increases with the commit time (*e.g.*, in DB2 we can use the LSN of the commit record).

In addition, to reduce restart processing, the capture service is periodically updated with the Inflight_LSN maintained by the capture program. The Inflight_LSN is the minimum LSN of inflight transactions and is normally written by the capture program to a persistent "restart queue". In our design there is no need for a persistent restart queue. The Capture service tracks the latest Inflight_LSN in memory. Periodically, it sends a heartbeat to the Monitor service, piggybacking the $commitTS$ for the latest δ, and the Inflight_LSN value at the time of the latest δ. The Monitor service uses these in two ways. First, to reduce log record processing on crash recovery. Second, to track the rate at which log reader is making progress.

At subscription initiation and restart from crash, the Monitor service invokes the following function on



Log, with LSNs of 1 st and last (commit) record marked for each transaction

Deltas sent to target

Delta	commitTS	seqNum
d_1	23	1
d_2	32	2
d_3	40	3
d_4	51	4
d_5	52	5

Occasional Heartbeats to Monitor Svc

inFlightLSN	commitTS
19	23
47	51

Figure 2: An example of capture side behavior

the Capture Service.

/*** Send out transactional deltas on given communication channel, for transactions that committed after $minCommitTS$. Append a sequence number to each delta, starting at $startSeqNum$.

$startLSN$ is a pointer in the log for the log reader to start reading from (*i.e.*, caller guarantees that relevant transactions do not start before $startLSN$).
***/

```
void subscribe(communicationChannel, minCommitTS, startSeqNum, startLSN)
{
```

1. Ask capture program to start sending *transactional* δ*s*. As an optimization, tell log reader it can start reading from $startLSN$, and that we only need deltas for transactions that committed after $minCommitTS$.
2. `seqNum = startSeqNum;`
3. `for each` δ `do` // loop forever
4. `if` ($\delta.commitTS \leq minCommitTS$) `continue;`
5. `Append` $seqNum$ `to` δ`;`
6. $seqNum++$`;`
7. Send δ on the communicationChannel.
8. Periodically (in a separate thread), send a heartbeat with the latest $\delta.commitTS$ and $Inflight\_LSN$ to the Monitor service. This is purely a performance optimization (to speed up crash recovery), so it is okay to do this infrequently.

}

Example: Figure 2 shows an example with 5 transactions, and the corresponding deltas sent by the capture service. In this example the capture side sends out only two heartbeats to the Monitor service. (although it could have sent out as many as it wanted). We will use this same example to illustrate apply functionality and crash recovery.

3.1.2 Apply Service

The Apply service maintains a transaction table *tranTbl* to keep track of deltas that have been applied to the target. *tranTbl* is a part of the target database, and contains an entry $< seqNum, commitTS >$ for each

transaction that has been applied (this table can be pruned significantly as we show below). The $commitTS$ is the commit timestamp of the transaction, and is used to identify whether a given delta has already been applied to the target. The $seqNum$ is the sequence number assigned by the Capture service. It is used for pruning, and to detect loss of deltas in transport.

To avoid checking this disk transaction table, the apply service loads a version of it into memory at subscription initiation and crash recovery. This is called the $inMemTranTbl$.

Definition [Max Contiguous Sequence Number]: Let $minSN = \min(inMemTranTbl.seqNum)$. Then, $maxContigSeqNum$ is defined as the highest sequence number such that $\{minSN, minSN+1, minSN+2, \ldots, maxContigSeqNum\} \subseteq inMemTranTbl.seqNum$.

The semantics of $maxContigSeqNum$ is that it is the highest sequence number received from the source since the last crash of the apply program. Therefore if the sequence number of a new delta is $> maxContigSeqNum + 1$ then we know a delta was dropped or delivered out-of-order.

Definition [Min Unapplied Sequence Number]: Let $minSN = \min(tranTbl.seqNum)$. Then, $minUnAppliedSeqNum$ is defined as the lowest sequence number such that $minSN < minUnAppliedSeqNum$ and $minUnAppliedSeqNum \notin tranTbl.seqNum$.

The semantics of $minUnAppliedSeqNum$ is that it is the earliest sequence number corresponding to transactions that have not been applied. $minUnAppliedSeqNum$ is calculated only while pruning the transaction tables (see `prune()` routine below). To reduce contention during this operation, we can calculate it approximately by reading $tranTbl$ into memory in "read committed" mode. This approximation will only reduce the extent of pruning and will not affect correctness.

The Monitor service initiates subscriptions and does crash recovery by invoking the following function on the Apply Service.

```
void subscribe(communicationChannel, tranTbl) {
```

1. Read $tranTbl$ from disk into an $inMemTranTbl$.
2. Periodically do `prune()` to reduce size of $tranTbl$ and $inMemTranTbl$.
3. Periodically send heartbeat to Monitor Service, piggybacking $\min(inMemTranTbl.commitTS)$. This is the time stamp up to which we know for sure all committed transactions have been applied at the target.
4. `for each` δ `do` // loop forever
    1. /* delta already applied, and entry is in transaction table */
       `if` $(\delta.commitTS \in inMemTranTbl.commitTS)$ `continue;`
    2. /* delta already applied, but corresponding entry has been pruned */
       `if` $(\delta.commitTS \leq \min(inMemTranTbl.commitTS))$ `continue;`
    3. /* channel has dropped or reordered a delta */
       `if` $(\delta.seqNum > 1 + inMemTranTbl.maxContigSeqNum)$ {
        - Ask channel to clean up, and ask the capture service to start sending from $1 + inMemTranTbl.maxContigSeqNum$.
        - `continue`

       }
    4. Add $< \delta.seqNum, \delta.commitTS >$ to $inMemTranTbl$
    5. Append the statement
       INSERT into tranTbl VALUES $\{\delta.seqNum, \delta.commitTS\}$ to the SQL for the δ. This ensures that $tranTbl$ will be updated atomically with the δ commit.
    6. Send δ to the target's apply program. The apply program is responsible for parallelizing applies through multiple apply agents.

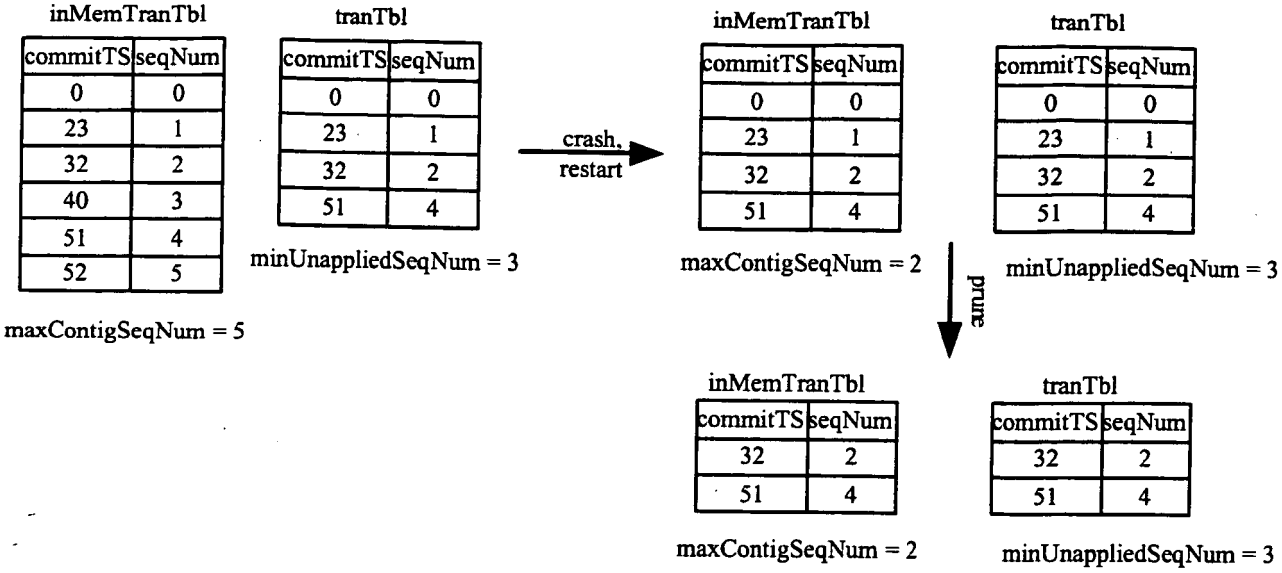


Figure 3: Apply side behavior for the example of Figure 2

}

/*** Prune $inMemTranTbl$ and $tranTbl$ of rows corresponding to most transactions that have been applied at the target. We only keep around the rows corresponding to unapplied transactions, and to transactions that were applied out-of-order (due to parallelism in apply). Effectively, the following invariants are maintained:

- $tranTbl \subseteq inMemTranTbl$
- All transactions with $commitTS \leq \min(inMemTranTbl.commitTS)$ have been applied.
- All transactions with $commitTS \in tranTbl.commitTS$ have been applied.

Thus after pruning, the number of rows in these tables is of the order of the number of parallel apply agents. ***/

`void prune() {`

1. Delete from $tranTbl$ and $inMemTranTbl$ all rows with $seqNum < minUnAppliedSeqNum - 1$.

}

Example: Figure 3 illustrates what happens at the apply side for the situation corresponding to Figure 2. The apply service accepts all five deltas, but only three are applied before it crashes. The applied ones are tracked in $tranTbl$ and loaded into $inMemTranTbl$ after the crash. They are then pruned.

3.1.3 Monitor Service

When the Monitor service gets a new subscription request, it performs the following routine.

`void subscribe(source, target) {`

1. Enter details of subscription into local persistent subscription table.
2. Initiate capture and apply service if needed.
3. Create $tranTbl$ at target, initialized with a single row $< 0, 0 >$. This signifies the beginning of replication.

4. Setup channel between capture and apply service. This need not be a persistent channel, and reliable delivery can be a best-effort guarantee (like TCP).
5. `ApplyService.subscribe(channel,tranTbl);`
6. `CaptureService.subscribe(channel,0,1,0);`

}

After this initiation, the Monitor Service keeps checking heartbeats from source and target to detect crashes. It also maintains a persistent table LogReaderProgress to track the progress of the log reader. Each entry in the LogReaderProgress is a pair $< inFlightLSN, commitTS >$, with the following semantics. At some point, the log reader read a transaction that committed at $commitTS$, and sent the delta to the Capture service. $inFlightLSN$ is less than or equal to the earliest log record of all inflight transactions at that point. Therefore $inFlightLSN$ is used to minimize log reads on restart.

`void handleHeartBeat() {`

1. `for each heartbeat from capture service do` // loop forever
   - If heartbeat times out, call `handleCrash(source, target);`
   - Append $< heartbeat.inFlightLSN, heartbeat.commitTS >$ to LogReaderProgress
2. `for each heartbeat from apply service do` // loop forever
   1. If heartbeat times out, call `handleCrash(source, target);`
   2. /* Prune the LogReaderProgress by removing all LSNs, except the LSN from where the log reader has to read if it crashes */
      - Let $floorTS$ be the maximum timestamp in LogReaderProgress.commitTS such that $floorTS \leq heartbeat.commitTS$.
      - Remove all rows from LogReaderProgress with $commitTS < floorTS$.

}

When a crash occurs the following steps are taken

`void handleCrash(source, target) {`

1. Restart apply service if it crashed.
2. Restart capture service if it crashed.
3. Setup channel between capture and apply service.
4. `ApplyService.subscribe(channel,tranTbl);`
5. /* Find the point upto which we know deltas have been applied */
   `lastSeqNum = min(ApplyService.inMemTranTbl.seqNum).`
   `lastCommitTS = min(ApplyService.inMemTranTbl.commitTS).`
6. Let $floorTS$ be the maximum timestamp in LogReaderProgress.commitTS such that $floorTS \leq lastCommitTS$. Let $startLSN$ be the inFlightLSN corresponding to $floorTS$.
7. `CaptureService.subscribe(channel,lastCommitTS,lastSeqNum+1,startLSN);`

}

Example: In the example of Figures 2 and 3, the LogReaderProgress table has two rows: $\{< 19, 23 >$ and $< 47, 51 >\}$ at the time of the apply side crash. After restarting the apply service, $lastSeqNum$ and $lastCommitTS$ are derived as 2 and 32 respectively. $startLSN$ is calculated as 19 (from the LogReaderProgress entries) and passed to the Capture service as the point to start reading the log from. Note that this recovery will work even if the Capture service had also crashed.

Further Optimizations: The Monitor service can also perform other optimizations to speed up restart. For example, the entire $tranTbl.commitTS$ at the target could be sent to the capture side, rather than only $lastCommitTS$. This will allow the capture service to know exactly which deltas have been applied at the

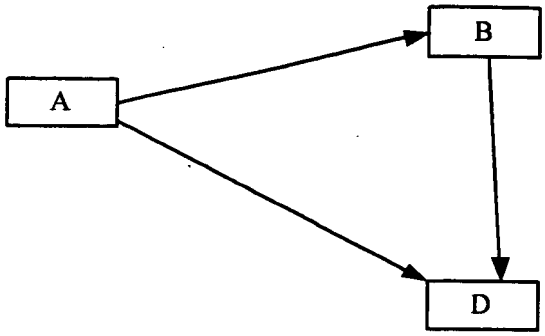


Figure 4: Update Anywhere topology

target, and avoid resending even a single one.

3.2 No Update-Anywhere, Single Source to Multiple Targets

When multiple targets are subscribing to the same source, each subscription is associated with a *subscription thread* in the capture service.

The Capture service functionality described in the previous section is easily extended to this situation. The capture service takes deltas from the source and puts them into a capture buffer (see Figure 1). Each subscription thread continually reads deltas from this buffer, assigns a sequence number, and sends them to the appropriate target. Periodically the capture service flushes from this buffer deltas that have been handled by all subscriptions.

The Monitor service's crash recovery role changes slightly. It needs to choose the $startLSN$ to be the *minimum* of the $startLSN$s required for each subscribing target. If the delta for this LSN happens to still be in the capture buffer (because the capture service may not have crashed), the log reade is not affected at all – otherwise it must start reading from this LSN.

In the grid context, some targets might be significantly slower than others, and might even have intermittent connectivity to the source. We do not want the capture buffer to grow arbitrarily large holding deltas for slow subscriptions. Therefore the capture service periodically "kicks out" subscriptions that are lagging behind too much. When the target comes back it resubscribes – at this point the capture service could either get the missing deltas from the database log, or do a full backup to bring the target up-to-date.

3.3 Update-Anywhere case

We assume that conflict resolution will be done within the apply program at the target. However, deltas could be duplicated within the replication dataflow, and we need to remove them explicitly. For example, in Figure 4, replication is setup from A to B, B to D, and A to D. Therefore an update at A will arrive at D twice, once directly and once via B. To avoid such duplicates we also tag each delta with its *birthplace* – the source where it originated, and its *birthplace commitTS*. When the apply service gets a delta it checks these two against its transaction table.